University of Bahrain
College of Information Technology
Department of Computer Science
First Semester, 2015-2016

ITCS215 (Data Structures)

Test II

Date: #/##/2015 Time: 16:00 - 17:15

STUDENT NAME (Uppercase characters)

STUDENT ID#	**2**	**0**						

SECTION#	

NOTE: THERE ARE SIX **(5) PAGES** IN THIS TEST

ONLY ONE SOLUTION WILL BE CONSIDERED FOR EACH QUESTION

QUESTION#	MARKS		COMMENTS
1	12		
2	18		
3	10		
TOTAL	40		

Question 1 [12 Marks]

Write a private member function called **insertItem** to be included in class **doublyLinkedList** that accepts two parameters. The first parameter is a pointer **ptr** to a node in the liked list. The second parameter is **item** of type **Type**. The function insert a new node having **item** as the *info* of the the node,int the list as follows:

- If the list is empty, create a doubly linked list of one node having **item** as the *info* of the node and make pointer **ptr** point to this node.
- If the list is not empty, then if the **item** is less than the *info* of the node to which pointer **ptr** is pointing then insert **item** before the node to which pointer **ptr** is pointing, else insert **item** after this node. Also make pointer **ptr** point to the new node inserted in the list.

Function Prototype:

```
    void insertItem(nodeType<Type>& *ptr, const Type& item);
```

Do not call any member function of class doublyLinkedList in your member function:

Assume that the class **doublyLinkedList** contains following private data member:

```
nodeType<Type> *first;      // pointer to the first node
nodeType<Type> *last;       // pointer to the last node
int count;                  // number of nodes
```

Also, assume that the struct `nodeType` is defined as follows:

```
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *next;      // pointer to the next node
    nodeType<Type> *back;      // pointer to the previous node
};
```

```
template <class Type>
void doublyLinkedList<Type>::insetItem(nodeType<ype>& *ptr, const Type &item)
{
   nodeType<Type> *newNode;
   newNode=new nodeType<Type>;
   assert(newNode!=NULL);
   newNode->info=item;
   newNode->next=NULL;
   newNode->back=NULL;
    if(count==0)
    {
      first=newNode;
      last=newNode;
      ptr=newNode;
      count++;
    }
    else
    {
      if(item < ptr->info)
      {
         ptr->back->next=newNode;
         newNode->next=ptr;
         newNode->back=ptr->back;
         ptr->back=newNode;
         ptr=newNode;
      }
      else
      {
         newNode->next=ptr->next;
         ptr->next=newNode;
         ptr=newNode;
         newNode->back=ptr;
         ptr->next->back=newNode;
      }
      count++;
    }
}
```

Question 2 [12 + 6 Marks]

(A) [10 Marks] Write a non-member function called **swapHalfStacks** that accepts an object **st** type **stackType** as parameter. The function swaps the first half of the stack with the second half of the stack. If the stack **st** is empty or contains odd number of elements, then do not do any swapping and return false, else return true after swapping.
Use only common stack operation such as `push, pop, top, isEmptyStack, isFullStack, operator=,` and `copy constructor.`
You can create local objects of type **stackType** in your function. Do not use array or any other data structure.
Example:
Before function call:

```
st:   5    17    10    15    20    18
      top
```

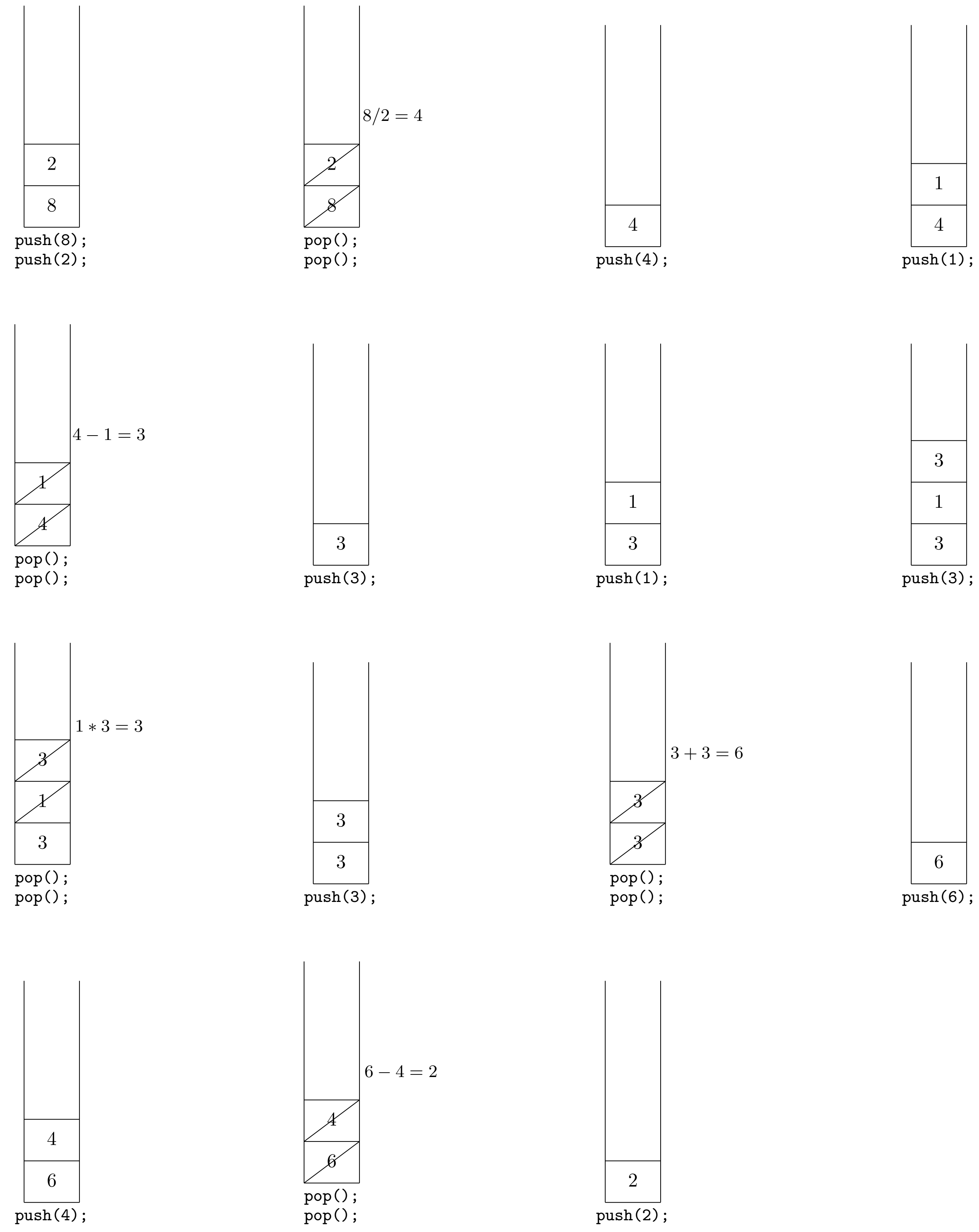
After function call:

```
st:   15   20    18    5     17    10
      top
```

```
template <class Type>
bool swapHalfStacks(stackType<Type>& st)
{
   if(st.isEmptyStack())
      return false;
   else{
          stackType<Type> st1,(st);
          stackType<Type> st2,st3;
          int count=0;
          while(!st.isEmptyStack())
          {
             count++;
             st.pop();
          }
       }
          if(count%2==0)
            return false;
          else{
                  for (int i=0; i < count/2; i++)
                    {
                      st2.push(st1.Top());
                      st1.pop();
                    }
                  for (int j=0; j < count/2; j++)
                    {
                      st3.push(st1.Top());
                      st1.pop();
                    }
                 while(!st2.EmptyStack())
                 {
                    st.push(st2.Top());
                    st2.pop();
                 }
                 while(!st3.EmptyStack())
                 {
                    st.push(st3.Top());
                    st3.pop();
                 }
              }
      return true;
}
```

(B) [6 Marks] Consider the following postfix expression. Use stack to evaluate it and show all the push and pop operations by clearly drawing the stack status.

8 2 / 1 − 1 3 * + 4 -



Question 3 [10 Marks]

Write a member function called **addWithoutDuplicates** to be included in class **queueType** that accepts an **item** of type **Type** as parameter and insert the **item** at rear of the queue, if the **item** is not already in the queue.

Function prototype:

```
void addWithoutDuplicates(Type& item);
```

Assume that the class has following data member:

`list:`	is the array
`maxQueueSize:`	the array size
`queueFront:`	index of the front element of the queue in the array
`queueRear:`	index of the rear element of the queue in the array
`count:`	the number of elements in the queue

Do not call any member function of class queueType in your member function:

```
template <class Type>
void queueType<Type>::addWithoutDuplicates(Type &item)
{
  if(count!=maxQueueSize)
   {
     bool found=false;
     int index=queueFront;
     for (int i=0; i<count; i++)
       {
          if (list[index]==item)
            {
              found=true;
              break;
            }
         index=(index+1) % maxQueueSize;
       }
      if(!found)
        {
           queueRear=(queueRear+1) % maxQueueSize;
           list[queueRear]=item;
           count++;
        }
   }
  else
     cout<<"queue is full";
}
```